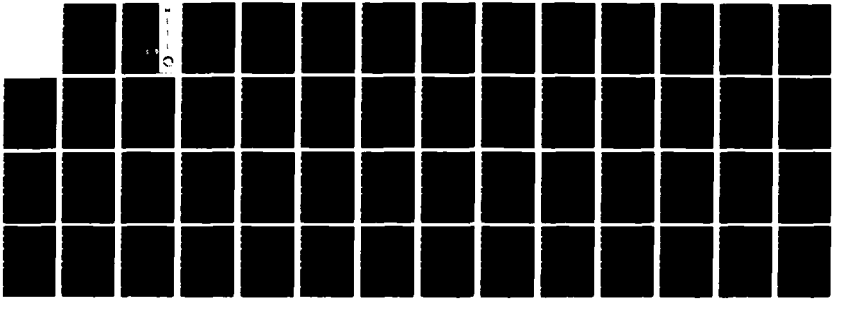
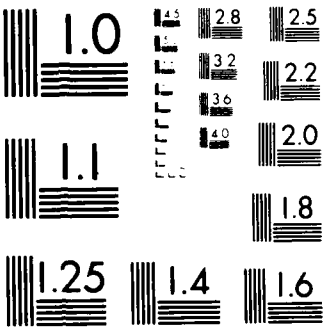
AD-A186 928 AUTOMATED ROUTE FINDER FOR MULTIPLE TANK COLUMNS(U) 1/1
ARMY ENGINEER TOPOGRAPHIC LABS FORT BEEVOIR VA
J R BENTON SEP 87 ETL-0480

UNCLASSIFIED F/G 15/6 ML





Automated route finder for multiple tank columns

John R. Benton

September 1987

Prepared for
U.S. ARMY CORPS OF ENGINEERS
ENGINEER TOPOGRAPHIC LABORATORIES
FORT BELVOIR, VIRGINIA 22060-5546

Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S) ETL-0480
6a NAME OF PERFORMING ORGANIZATION U.S. Army Engineer Topographic Laboratories 6b OFFICE SYMBOL (*If applicable*)	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (*City, State, and ZIP Code*) Fort Belvoir, VA 22060-5546	7b ADDRESS (*City, State, and ZIP Code*)
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61101	A91	D	01

11 TITLE (*Include Security Classification*)
Automated Route Finder for Multiple Tank Columns

12 PERSONAL AUTHOR(S) John R. Benton

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
Technical	FROM _____ TO _____	September 1987	

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The Automated Route Finder for Multiple Tank Columns computes multiple non-competing paths for columns of tanks. The network of available paths is represented by a graph-theoretic structure. Each arc of the graph has an associated cost which represents the time required to traverse the path corresponding to the arc. A best-first algorithm is used to search the graph in order to find the specified number of optimum paths. The algorithm was implemented on the Symbolics LISP Machine with a color monitor used to display the graph as it is explored. Sample outputs of route finding are included with an analysis of the results. Future enhancements for the system are outlined.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (*Include Area Code*)	22c OFFICE SYMBOL
E. James Books	(202) 355-3039	ETL-IM-T

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PREFACE

This study was conducted under DA project 4A161101A91D, work unit 01, "Automated Route Finder for Multiple Tank Columns."

The work was done during fiscal year 1985 under the supervision of Anne Werkheiser, Team Leader, Center for Artificial Intelligence, and Robert D. Leighty, Director, Research Institute.

Col Alan L. Laubscher, CE is Commander and Director, and Mr. Walter E. Boge is Technical Director of the U.S. Army Engineer Topographic Laboratories during the report preparation.

TABLE OF CONTENTS

I. INTRODUCTION 3

II. ANALYSIS 4

A. Multiple Route Finder Program 6

B. Graph Generation Using Mouse 11

C. Line and Curve Drawing Functions 12

D. Screen Control Functions 12

E. Special Macros 12

III. CURRENT STATUS 14

IV. FUTURE ENHANCEMENTS 19

V. REFERENCES 20

VI. APPENDIXES 21

A. Multiple Route Finder Program 22

B. Graph Generation Using Mouse 35

C. Screen Control Functions 39

D. Line- and Curve-Drawing Functions 44

E. Special Macros 47

F. Discussion of Avenues of Approach and Obstacles 49

LIST OF ILLUSTRATIONS

Figure 1. Examples of line-thinning algorithm used to generate stick figures 5

Figure 2. (a) Graph representation of routes and (b) Corresponding coincidence matrix 8

Figure 3. Search tree with start node of 1 and goal node of 5 for graph of Figure 2 9

Figure 4. List of global variables used in multiple route finder 10

Figure 5. Flow chart of principal functions in the multiple route finder program 11

Figure 6. Sample output of multiple route finder program 15

I. INTRODUCTION

Anyone who has struggled to find the most direct a route between two locations on a city map where there is no single road or even just two or three roads directly connecting the two locations is familiar with the general problem of planning an optimum path through a complicated tangle of roads. In planning a route through a city one must consider not only distance but also the number of traffic lights and quantity of traffic at a given time of day. A battlefield commander may not have to worry about traffic lights when planning a route, but he must consider positions of enemy and friendly forces, lines of sight from enemy observation points, availability of places of concealment from aerial observation, choke points where his forces could be ambushed, and weather conditions that could affect the mobility of his vehicles.

Currently, there is no capability of automated route finding available for operational use in the Army. Such a capability would be particularly useful in situations that require a rapid reaction to counter an enemy threat or changing conditions. One example of an Army application where this capability would be valuable is the requirement that a friendly forces commander be able to predict the enemy's objective and the routes the enemy will use to advance toward their objective. Military doctrine often calls for offensive forces to advance in three separate columns. The defending forces must use their knowledge of the doctrine and tactics of the opposing force as well as knowledge of the terrain to anticipate and possibly to counteract the movement of their enemy. The entire Intelligence Preparation of the Battlefield (IPB) process is now done manually, and depending on the size of the area, requires as much as two or three months to complete. In order to have the capability to develop an IPB in response to enemy actions it will be necessary to automate most of the required effort. One of the tasks that could be performed by the computer system would be to predict enemy movements. As mentioned above, frequently the military plans will involve three separate columns. The computer should therefore compute three optimum noncompeting paths as well as suboptimum alternatives. The three routes must be noncompetitive (i.e. have no road segments in common) since traffic jams would be engendered if two separate columns had to share the same road.

To address these problems, a project was initiated to develop a system capable of automatically generating optimized routes for multiple tank columns. Input to the current program is a graph-theoretic representation of the road network. A pictorial representation is displayed on a television monitor. Optimum paths are computed by using the required traversal time for a road as the cost of traversing the road. As the paths are computed they are displayed as a color overlay on the monitor and also are printed as an ordered list of graph nodes.

II. ANALYSIS

All of the information required to determine the paths to an objective may be present in a cross-country mobility map. However, this does not mean that it will be easy for even a trained analyst to quickly determine multiple optimum routes. Since the allowable velocity along a selected path can vary according to the type of terrain encountered, the only way that the traversal time can be computed is to integrate the velocity along the path.

In developing a computer model for route planning over a large area, a prime concern must be to avoid what is known as "combinatorial explosion". For example, a four-levels-deep full binary tree has 31 nodes while a ten-levels-deep full binary tree has over two million branch points or nodes. If the map data consist of a grid at ten meter spacings, an unrestricted search algorithm would generate a search tree with literally millions of nodes* in order to plan a route for a distance of 100 meters. Obviously, no one would actually use such a simple-minded technique, but when route planning over a distance of several kilometers is required, the almost inevitable result of planning a route at the pixel level is combinatorial explosion. An alternative approach is to reduce the size of the search space before the planning algorithm is invoked.

There are several methods that can be used to reduce the search space. However, they can be divided into two classes: (1) preprocessing methods and (2) methods for searching more intelligently. The preprocessing methods are concerned with the data representation of a mobility map. The maps are normally stored in one of two forms in a computer, raster and vector. The raster is simply the ordered rows of pixels mentioned above. The vector representation makes use of the fact that typically a large area or the map has identical mobility factors. This area can be at least approximately bounded by a polygon of contiguous vectors. The area, or "region" as it is generally called, can then be represented in the computer by a list of the vectors of which the polygon is composed. At the pixel level, each pixel is surrounded by either four or eight neighboring pixels, depending on the definition of neighbor that is used. Thus from a given node the number of neighbors and the distance to each neighbor are always constant. In contrast, with a vector representation, the number of neighbors will vary, and the distance to each neighbor is no longer well defined since the shape of the the regions may be irregular. A method intermediate in storage efficiency uses a quadtree representation[1] in which the map of a square area is divided into four quadrants. Each quadrant is subdivided if the quadrant is not completely contained within a uniform region. The process terminates when no regions remain to be subdivided. The distance from the center of any quadrant to the center of an adjacent quadrant (which may have been subdivided a different number of times and therefore is a different size) is easily computed.

There is one additional representation method which can result in a large reduction in storage requirements. Any one of a number of line-thinning algorithms can be applied to the mobility map in order to generate a skeletal structure that will correspond to a line drawn along the center of mobility corridors or avenues of approach.† Branches in the skeletal structure are caused by obstacles that force a traveler to fork either to the left or to the right in order to go around the obstacle. Since the line-thinning algorithms require a binary image, the first step is to generate a mobility map containing only two levels of mobility: go or no-go. An illustration of this technique is shown in figure 1.[2] The three shaded figures represent regions, with the corresponding skeletons indicated by the superimposed bold characters. These skeletons can also be considered to be graph-theoretic structures to which graph theory can be applied. A tree-searching algorithm can be used to find the optimum paths through the graph.

* A node indicates a point along a path where there is a choice in which direction to proceed.

[1] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Surveys* **16**, 2 (1984).

† See Appendix F for a discussion of avenues of approach and obstacles.

[2] Zhang, T. Y. and Suen, C. Y., "A Fast Parallel Algorithm for Thinning Digital Patterns," *Comm. of the ACM* **27**, 8 (1984) 236-239.

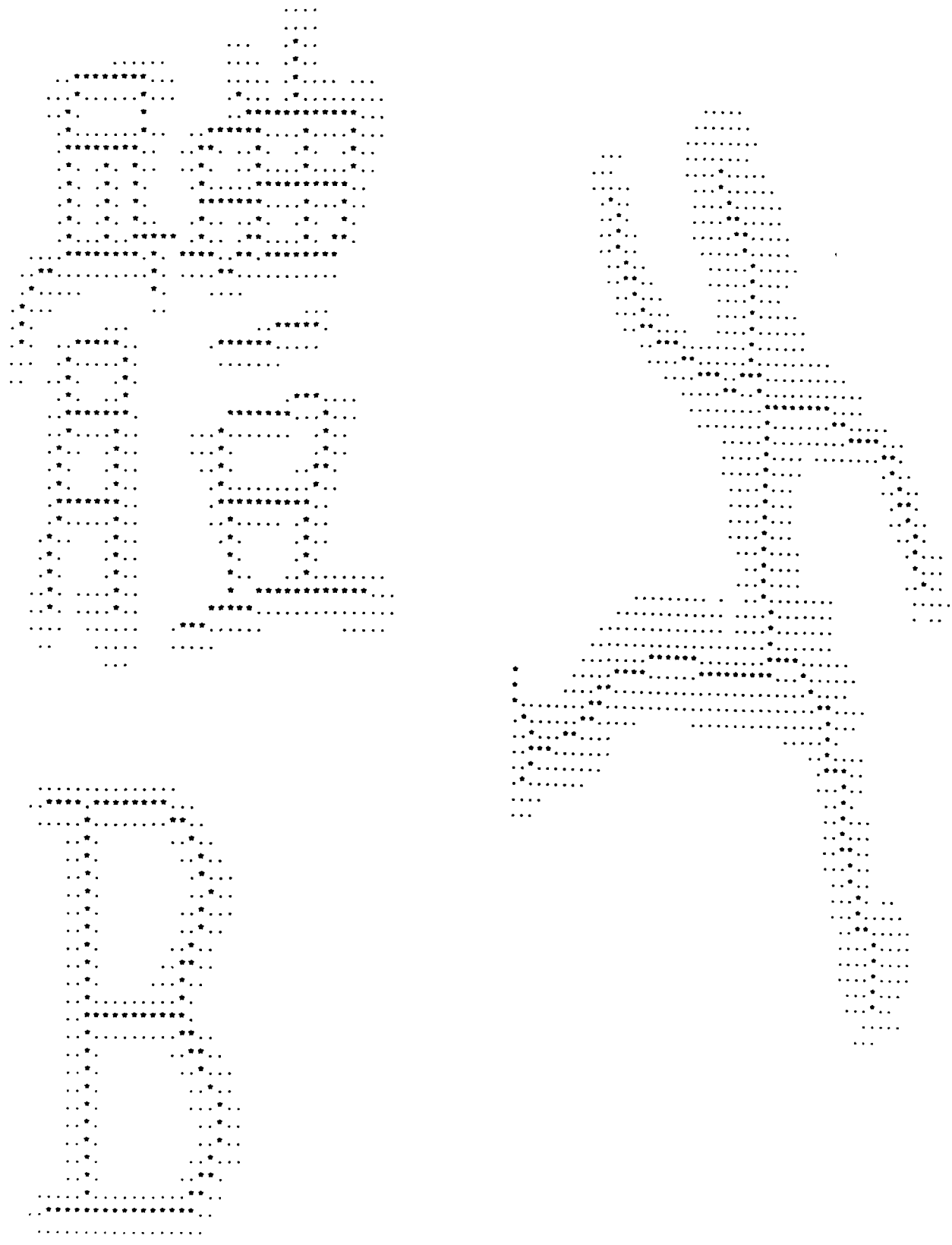


Figure 1. Examples of Line Thinning Algorithm Used to Generate Stick Figures

The approach outlined above was chosen for use on this project. Other groups are currently working on the problem of reducing trafficability maps to graphs. Therefore, effort was concentrated on generating multiple routes rather than first developing a graph generation module. The following sections describe the software developed in support of this project. The reader is assumed to have a working knowledge of the LISP programming language.

A. Multiple Route Finder (MRF) Program

The input to this program is a graph representation of the skeleton of a mobility map as described in the previous section. The user enters the desired start and destination points and the desired number of routes connecting the two points. The program computes the paths and prints the computed routes on the monitor.

Initially the graph is represented by a coincidence matrix, with a cost associated with each arc of the graph. The (m,n) component of a coincidence matrix is the cost to traverse the path between the m and n nodes. If there is no path between any two given nodes, the cost is represented as -1. (The actual cost should be infinity, but -1 is easier to represent in the computer.) The cost of n,n is of course zero, and the diagonal of the matrix therefore contains zeroes. Since it is assumed that the cost in going from m to n is the same as the cost in going from n to m, the value of the matrix element m,n equals the value of the matrix element n,m. An NxN input matrix is immediately converted into a one-dimensional array of length N where each element of the array is a list of doublet values. For the nth element of the array, each point with a path to n is represented by a doublet. The doublet contains the node number m and the cost of traversal from n to the node m.

An algorithm essentially equivalent to the A* algorithm* developed by Hart, Nilsson, and Raphael* was used to search the graph structure for an ordered listing of noncompetitive optimum routes between two given nodes. Each route is represented by a linked list of nodes, with the first node being the start and the final node being the destination node. The route with the lowest cost is listed first. Since the routes are noncompetitive, no two routes can share a path and thus they cannot share two successive nodes in their respective paths. The program is currently more restrictive in that no two paths can share a road intersection. Depending on the road widths at intersections, this may or may not be a necessary restriction.

The major difference of the multiple route finder (MRF) algorithm used here from A* is that A* halts as soon as the first optimum route is discovered, while MRF keeps searching until either there are no more unexplored routes or the required number of routes has been found. A* keeps two separate lists of nodes called OPEN and CLOSED. OPEN is the list of nodes that have not yet been explored. Initially, it contains only the start node. When a node is explored, it is moved from OPEN to CLOSED, and its descendants are put on the OPEN list. However, if one of these nodes has previously been reached by another route, then the more costly of the two routes must be pruned, and only the cheaper of the two routes will be retained. The MRF algorithm differs from A* in that the OPEN and CLOSED list are kept as a single linked list, with a slot at the node address to specify whether the node type is TERMINAL (CLOSED), NONTERMINAL (OPEN), PRUNED, or DESTINATION. A* terminates when the destination node is reached, in contrast to MRF, which continues to search for additional routes; therefore an explicit DESTINATION type is needed.

The A* algorithm uses a heuristic function to estimate the cost to go from the current node to the destination node. This function is usually called h(n), where n is the current node. The total estimated cost associated with the node n is f(n) = g(n) + h(n,) where g(n) is the actual cost in traveling from the start node to the node n. Thus, f(n) is the estimated total cost for

* A good description of the algorithm as well as A* in the literature is contained in the textbook by P. H. Winston, *Artificial Intelligence*, 2nd Ed., Addison-Wesley Publishing Company, 1984. More rigorous developments are in Nils J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980 and Judea Pearl, *Heuristics*, Addison-Wesley Publishing Company, 1984.

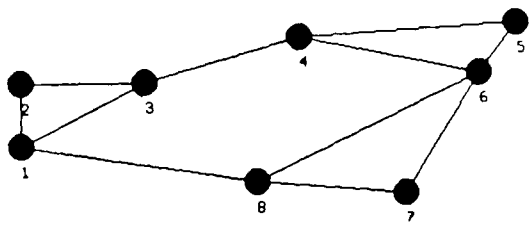
† Hart, P.E., Nilsson, N.J., and Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, SSC-4, 1968, 100-107.

traversing a path from the start node to the point n and then on to the destination node. A heuristic function can be shown to be "acceptable" if it is monotonically nondecreasing and nonnegative. If the heuristic function is acceptable, then the algorithm is guaranteed to terminate with the optimum solution. Conceptually, the simplest heuristic that meets these requirement is the euclidean distance from the the node n to the destination node. The actual cost can equal but never be less than this straight line distance from n to the destination.

The basic data structure used for this program is the defstruct, which is available in ZetaLisp and Common LISP. It is similar to the disembodied property list of FRANZ LISP. Each node defstruct has six slots containing ◦ node number ◦, ◦ father-pointer ◦, ◦ down-pointer ◦, ◦ weight ◦, ◦ f-weight ◦, ◦ type ◦, and ◦ coordinates ◦. The node numbers correspond to the number used in the coincidence matrix, the father pointers are used to represent the search tree, the down-pointers provide the links for the linked list of nodes, weight is the accumulated cost from the start node to the subject node, f-weight is the sum of weight and the estimated cost of reaching the destination node, type is as explained in the previous paragraph and coordinates are the geographical coordinates of the intersection represented by the node. The entire search tree can be explored by traveling along the linked list from the first node to the last node. At each DESTINATION node or TERMINAL node, trace the father pointers up to the start node. A graph and the corresponding coincidence matrix are shown in figure 2. The explanation of the search tree can be simplified by setting the heuristic function to zero. The distance along an arc connecting nodes m and n corresponds to the value in matrix element (m,n). The corresponding search tree is shown in figure 3. The nodes of the tree correspond to the nodes of the graph of figure 2(b). The starting point for the search tree is node 1 and the goal is node 5. There are three descendents of node 1. The corresponding distances from node 1 are shown adjacent to nodes 3, 2, and 8. Node 3, in turn, has only two descendent nodes since node 1 is an ancestor node. The cumulative distances from node 1 to nodes 2 and 4 are again adjacent to the nodes. At this point, it should be observed that another path to node 2 has already been discovered, but with a distance of only 300 compared to 1800. If the sub search trees for each of the identical nodes are expanded, the tree will be identical except that the weights on one tree will be 1500 greater (1800 - 300) than for the other tree. Therefore, the subtree with the larger weights can be pruned. This is indicated by the slash across the line and the letter P below the slash. The procedure outlined above is continued until either of two conditions has been satisfied: (1) all remaining terminal nodes have been pruned; or (2) the required number of routes has been found and all terminal nodes have either been pruned or have weights greater than any of the destination nodes.

Three accessor functions for the defstructs were written and are widely used throughout the program. These are (1) **new** , (2) **fetch** and (3) **pstore.** These functions were originally used to access property lists when the original version of this program was coded to run in Franz LISP. In the ZetaLisp implementation, it is possible to use either the property list version of these functions or the more efficient defstruct version. The function **new** generates a **gensym** name and interns it (i.e., puts the name on the *oblist*). The function **pstore** with arguments *name*, *value*, and, *property* is directly analogous to the **getprop** function. Similarly, **fetch** has arguments *name* and *property* and returns *value*.

Figure 4 is a list of all global variables and a list of every function along with a list of functions called by a given function. Figure 5 is a graphical representation of the function calls but with the low level accessor and graphics calls deleted in order to simplify the drawing. Recursive calls are shown to the functions **insert-answer** and **graph.** The function **get-graph** contains a sample coincidence matrix. In actual practice, a matrix would be read from a disk file. A listing of the program is in appendix A.



(a)

	1	2	3	4	5	6	7	8
1	0	300	1000					1700
2	300	0	800					
3	1000	800	0	800				
4			800	0	1500	1000		
5				1500	0	250		
6				1000	250	0	300	1400
7						300	0	1000
8	1700					1400	1000	0

(b)

Figure 2 (a) Graph Representation of Routes and (b) Corresponding Coincidence Matrix

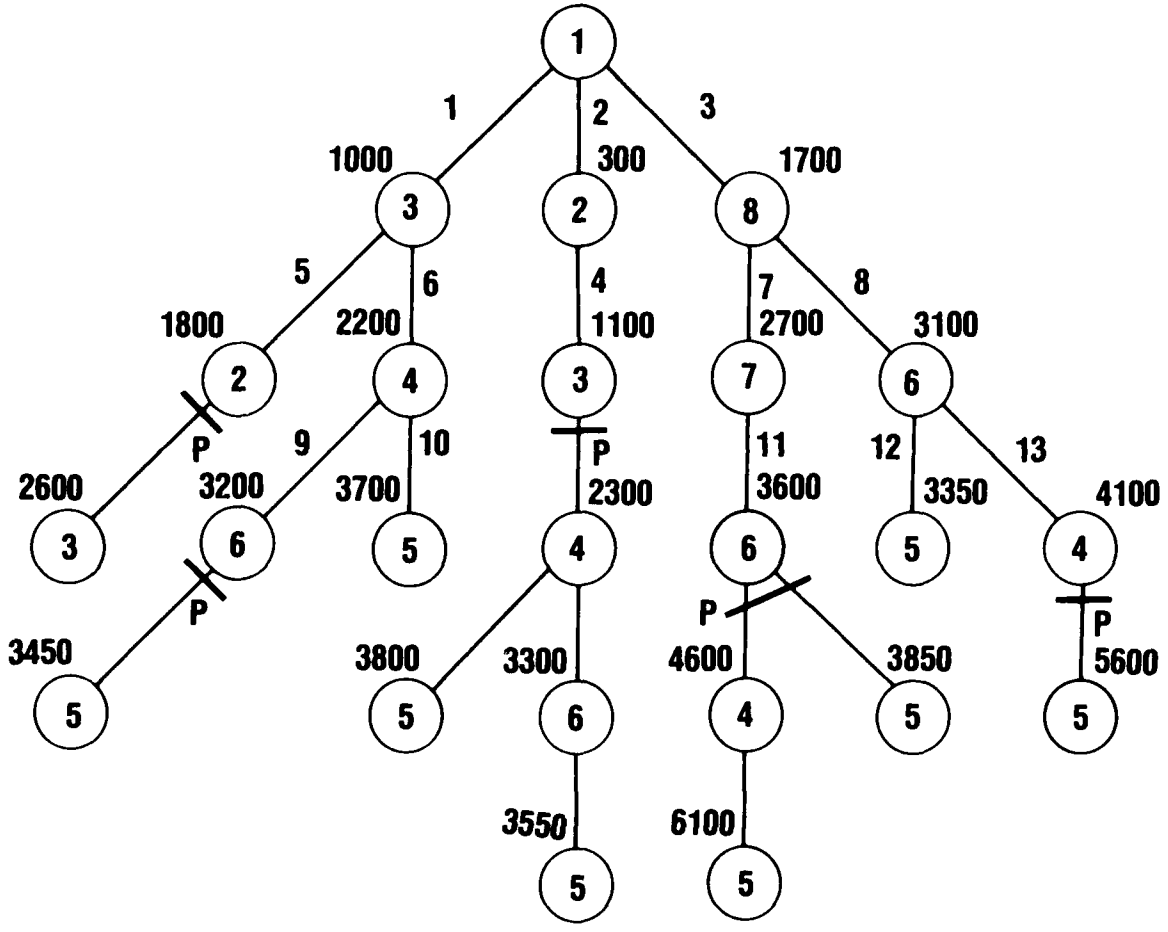


Figure 3. Search tree with start node of 1 and goal node of 5 for graph of Figure 2

List of functions and calls to other functions and global variables:

```
triple:
  get-graph define-window pcopy pstore draw-arc update checklist
  send print-answer new
  GLOBALS USED: no-nodes-left finished startnode destnode nbrpathsfound
            : answerlist top ptrnode1 node1
get-graph
  store-graph GLOBALS USED: coordinates data arr
store-graph
  graph
graph
  graph
new
fetch
update
  fetch pcopy pstore new draw-arc insert-node pop print-list
  GLOBALS USED: finished top ptrnode1
draw-arc
  fetch line
insert-node
  fetch store-soln pstore find-link GLOBALS USED: destnode
find-link
  fetch pstore prune-check
store-soln
  prune-check fetch pstore insert-answer
  GLOBALS USED: nbrpathsfound answerlist
insert-answer
  insert-answer
pull
  fetch pstore  GLOBALS USED: no-nodes-left finished top
prune-check
  fetch prune GLOBALS USED: destnode
prune
  fetch pstore
print-answer
  pop fetch
  GLOBALS USED: start-node dest-node
penult
  fetch
checklist
  pop fetch penult push
    GLOBALS USED: no-nodes-left finished startnode answerlist top
print-list
  fetch GLOBALS USED: top
make-colors
line
```

Figure 4. Global variables for multiple route follower

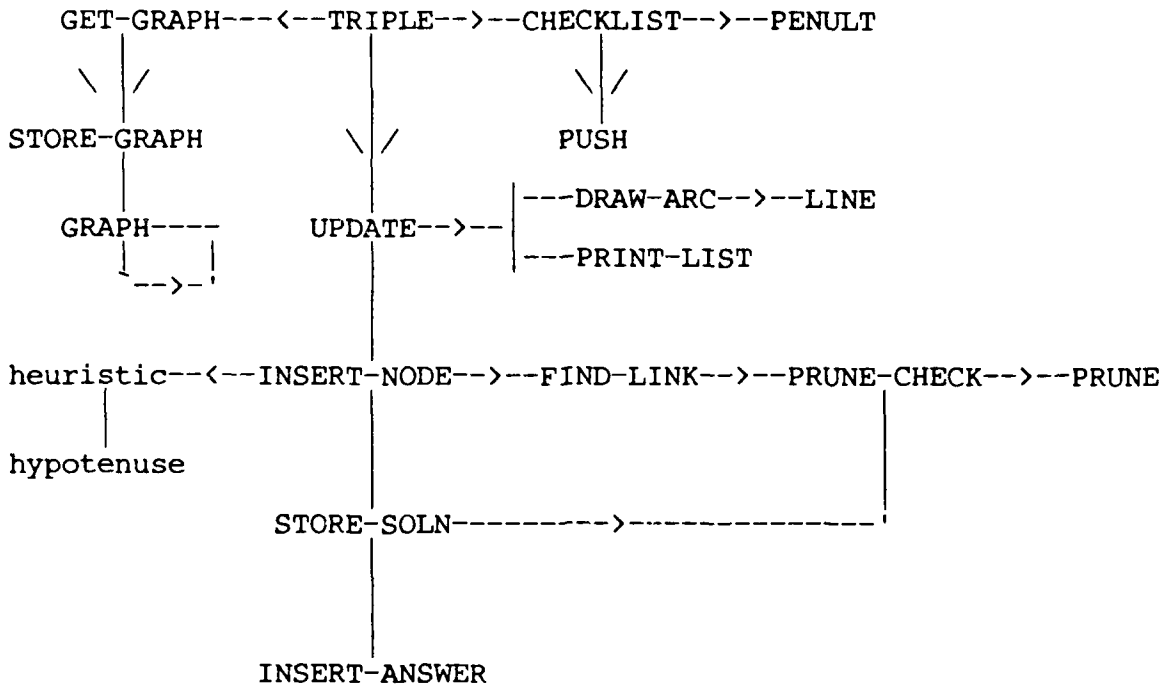


Figure 5. Flow Diagram of Principal Functions Used in MRF

B. Graph Generation Using Mouse *

This module provides a capability for the operator to manually input a graph representation of the mobility map by using the mouse device to trace mobility corridors on the color monitor. The software automatically generates the internal data representations. The top level function in this module is **get-graph-using-mouse.** The MRF module calls either **get-graph** or **get-graph-using-mouse** depending on whether graph data have been digitized previously or are to be obtained using the mouse to extract routes from a cross-country mobility map displayed on the color monitor. This function first initializes several arrays and then tests to see if a monochrome or a color window is to be created. The variable *win-1* is bound to the window name, and the global variable ***flag**** is set to true. The function **trace-arc-using-mouse** is in a loop that continues until nil is returned by the function.

* A mouse is a device that is used for controlling the cursor position. It consists of a small hand-held box with a small trackball mounted flush with the bottom of the box. When the box is slid over a smooth surface, the track ball can rotate in either of two dimensions. The mouse device also contains three buttons that the operator can push in order to signal the computer.

The function **trace-arc-using-mouse** calls **draw-segment,** which fits spline curves to a road segment. One or more segments constitutes an arc with a solid circle drawn at each end of the arc. When **draw-segment** returns nil, the last arc has been drawn. Otherwise, **draw-segment** returns a list with two atoms, which is then bound to the variable curve. The first value is the traversal distance of the segment, and the second value is used to test if the segment is the last segment of an arc. If it is, then the node at the end of the arc is drawn if it was not previously drawn. If the segment is the first segment of an arc and the function **old-node-p** returns nil, then the node at the beginning of the arc is drawn.

The function **old-node-p** is a predicate function that tests to see whether the coordinates for the new node correspond to any previously tagged node within a precision of 10 pixels for each axis. Thus, nil is returned if the node is new, and the node number is returned if the node corresponds to the location of a previously tagged node.

The function **plot-solid-circle** has four arguments: *x* and *y* coordinates of the circle, the node number, and the color of the circle. The function prints a solid circle, prints the node number in the center of the circle, and then repositions the cursor at center of the circle.

C. Line and Curve Drawing Functions

The function **draw-segment** draws lines on *win-1* by calling the function **traverse-road,** which in turn calls the function **connect-points-with-line.** Mouse clicks are used as follows: left click puts point, middle click indicates point is last point. The straight line curve connecting the points is cleared prior to drawing a cubic spline curve through the points with **draw-cubic-spline.** The number of points used to generate the curve is returned.

The function **traverse-road** calls **connect-points-with-line** and uses the Pythagorean theorem to compute the length of each straight-line section defined by the arrays x-cor and y-cor. The lengths are summed to get an approximate value for the length of the spline curve fitted to the sequence of points. If **connect-points-with-line** returns nil, then **traverse-road** will also return; otherwise, the segment distance will be computed and returned. Nil is returned by **connect-points-with-line** to signal that no more arcs remain to be traced.

The function **connect-lines-with-points** places points on a window with the mouse and draws lines between them while storing the point coordinates in two one-dimensional arrays. *Window* is the exposed window where the points are placed, and *x-cor* and *y-cor* are the two arrays where the relative window coordinates are stored. Clicking left once places points; clicking middle places the last point and exits. The number of points is stored in the *fill-pointer* for array *x-cor;* the function then returns the number of points.

D. Screen Control Functions

A global variable is used to specify whether a color or a monochrome screen will be used to display the graphics. The variable called **screen-type** can have two valid values, which are (1) color: a color window will be created on the color monitor, and (2) b-w: a monochrome window will be created on the right side of the monochrome monitor. Similarly, the function **draw-arc** will call either **bw-draw-arc** or **color-draw-arc** depending on the value of **screen-type**.

E. Special Macros

Two macros are used by the other software modules and are defined below The function **iff** is the standard

(if form1 then form2 form3 ... else form_a form_b ...)

construct. The key word "then" must be used, but "else" is optional.

The macro **loop1** has syntax similar to the **cond** function except that it loops instead of falling through if no predicate evaluates to t. The syntax is shown below:

```
(loop1
    ((fcn_1) form_1 form_2 form_3 ...)
    (fcn_2)
```

(fcn_3)

..

((fcn_n) form_A form_B ...))

where fcn_x indicates a function and form_x indicates an arbitrary LISP form or expression. If the macro sees only a single parenthesis at the start of a line, it does not treat the function that follows as a predicate, but simply executes the function and drops to the next line. Alternatively, ((setq a t) 'exit) will result in the loop being exited with a value of exit returned.

III. CURRENT STATUS

The software described in the previous section has been debugged and tested with a relatively limited set of data. Special cases that were believed to be most likely to cause problems were included in the test. Currently the graphics display shows the routes that have been explored, but neither annotates the selected paths from the start to the destination, nor shows the arcs that have been pruned. These capabilities will be added to the system in the near future. Tests will also be conducted using larger graph structures. Figure 6 shows the output of the multiple route finder for six cases using the graph of figure 3(a). Following the words "father is n" is the linked list of nodes that exist after inserting the descendants of "n" into the list. In the linked list, the node suffixes have the following meanings:

N	-	Nonterminal node
D	-	Destination node
P	-	Pruned node
none	-	Terminal (i.e., unexplored) node

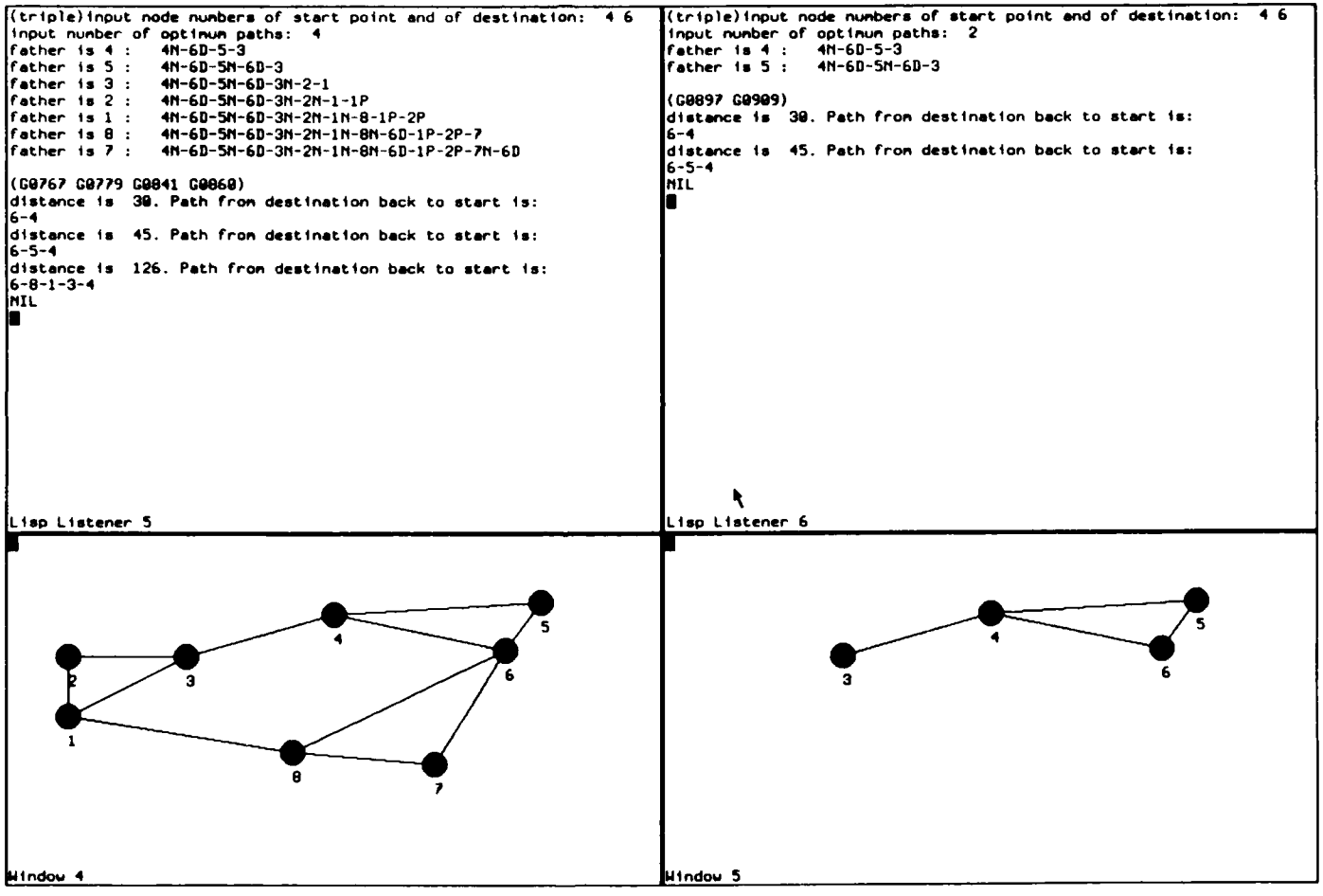
(1) Start node = 4 and destination node = 6, number of routes = 4.
Examining the output of case 1, one sees that node 4 is opened and its three siblings are added to the linked list "4N-6D-5-3". Node 4 is nonterminal because all its siblings are on the list and node 6 is the destination node. On each of the succeeding lines the left-most terminal node is opened and its siblings are inserted into the list. Finally, at the line "father is 7 :," the last unexplored node has been explored and four destination nodes are on the list; however, only three of these nodes represent legitimate paths. To understand why this happened, consider the previous line, where node 8 was opened, and nodes 6 and 7 were added to the list. Node 6 is immediately recognized as the destination node and is changed to type destination and added to the *answerlist*. Node 7 is now opened and its sibling node 6 is recognized as the destination node. The function **prune-check** fails to recognize that one of the two node-pointers to node 6 should be pruned because node 6 is the destination node. To solve this problem, the function **checklist** calls **penult** to see if any of the pointers on the *answerlist* have identical penultimate nodes. Since the paths are traversals from the destination node to the start node, the penultimate node of a search path is actually the second node in the forward direction. When two such paths are found, the longer path is deleted from the *answerlist*. This happened in case 1, and thus the path 6-8-1-3-4 was eliminated.

(2) Start node = 4 and destination node = 6, number of routes = 2.
In this case only two paths were requested, and the search was terminated when the two were found, since the only node open was 1 and the distance from node 4 to node 1 is greater than either of the paths on the *answerlist*.

(3) Start node = 4 and destination node = 6, number of routes = 1.
In this case the distance from node 4 to node 3 is less than the distance from node 4 to node 6. However, f(3) = g(3) + h(3) is greater than f(6) = g(6), and the one destination node is found immediately.

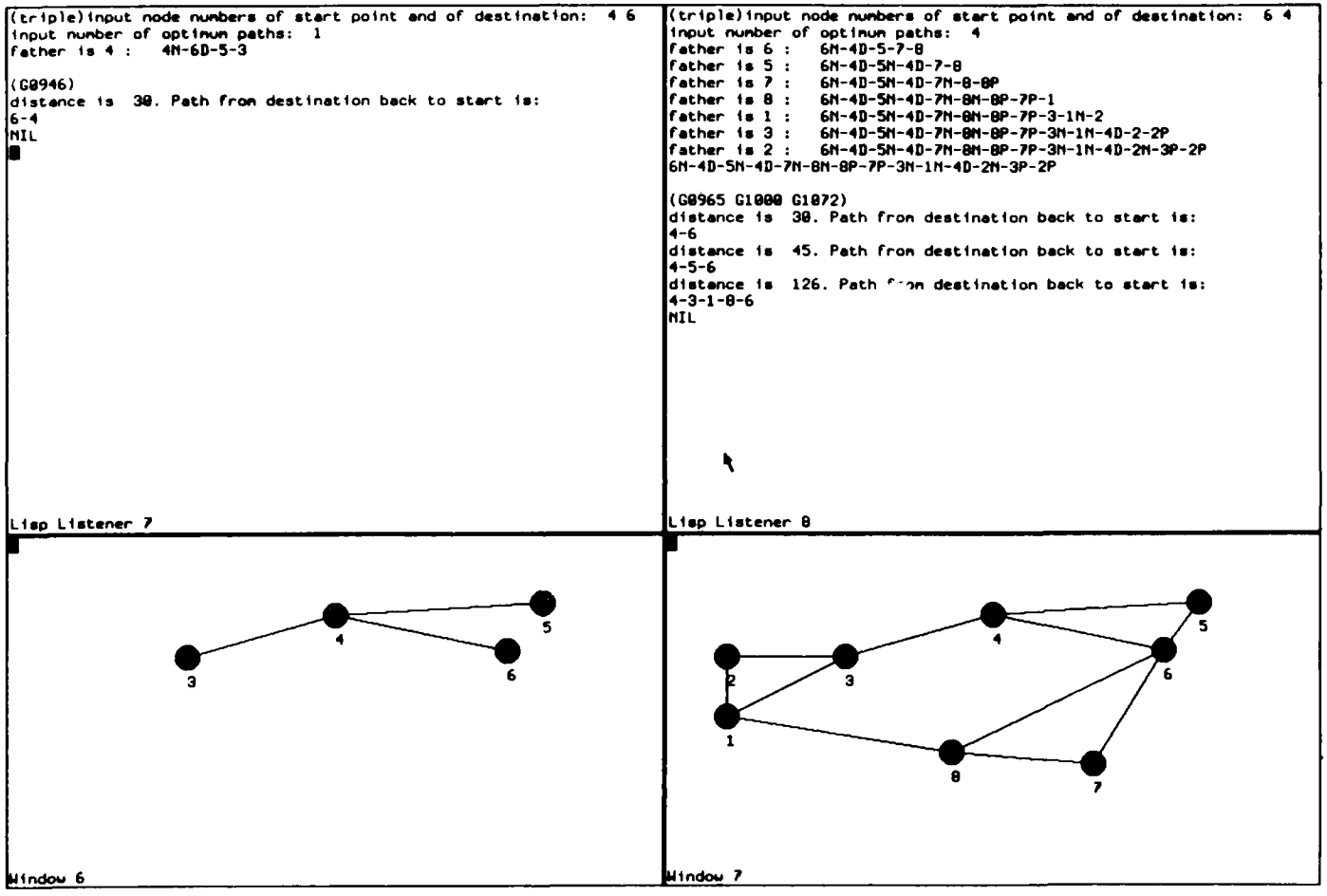
(4) Start node = 6 and destination node = 4, number of routes = 4.
This case is the same as case 1, except that the direction is reversed. As expected, the selected paths are simply in the reverse direction.

(5) Start node = 1 and destination node = 5, number of routes =3.
In this case the first optimum path computed was 5-6-4-3-1. The subpath 5-6-4 was chosen instead of 5-4 since it is shorter. However, inclusion of node 6, clobbers the alternative route 5-6-7-8-1. Ideally, MRF should have selected the longer route for the first path so that the second path could go through node 6. There are several possible approaches for solving this problem that



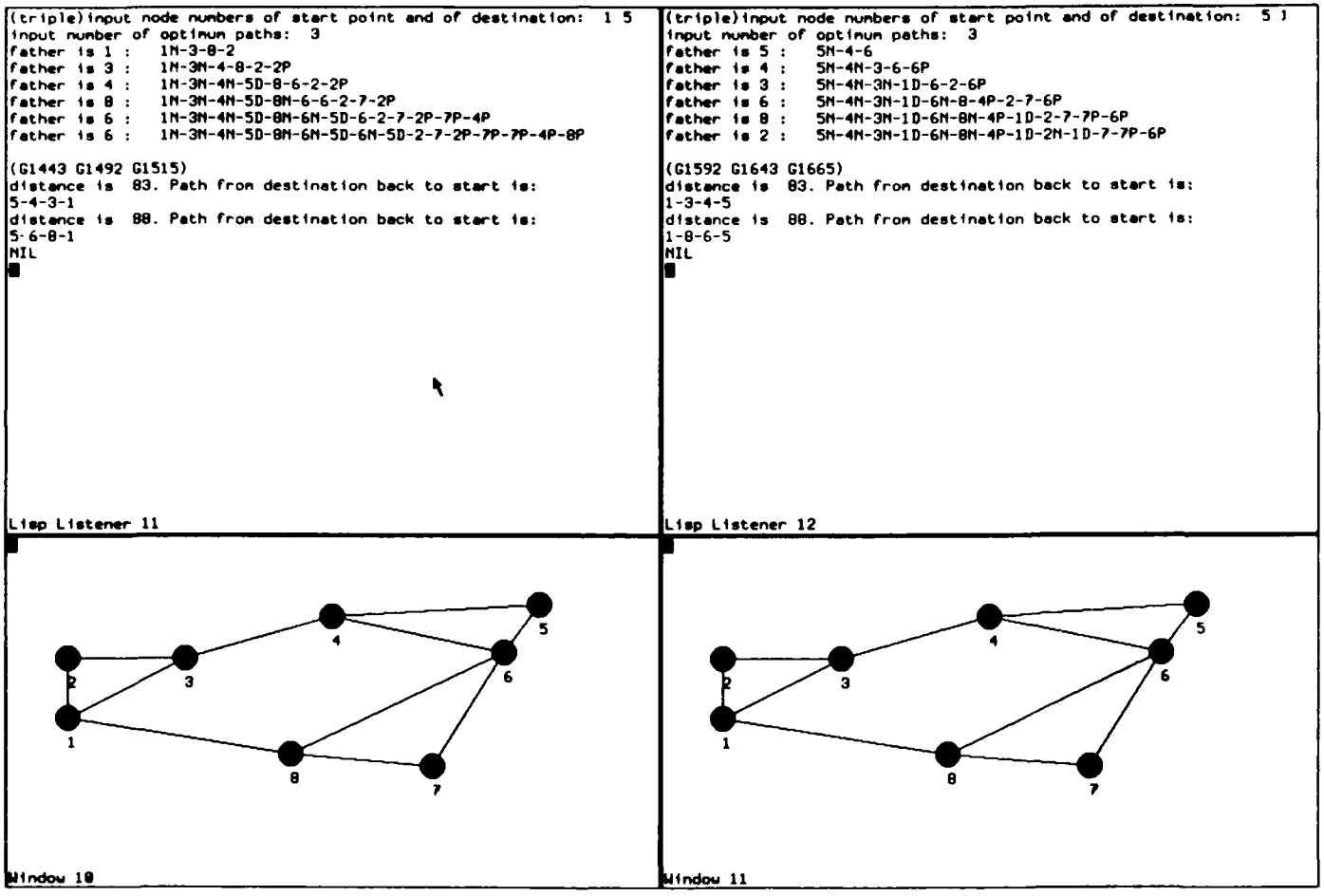
CASE 1 CASE 2

Figure 6. Sample Output of Multiple Route Finer Program



CASE 3 CASE 4

Figure 6. Sample Output of Multiple Route Finer Program, Continued



CASE 5 CASE 6

Figure 6. Sample Output of Multiple Route Finer Program, Continued

will be explored in future work on this project.

(6) Start node = 5 and destination node = 1, number of routes = 3.
This case is the same as case 5 except that the direction is reversed. As expected, the selected path is simply in the reverse direction.

IV. FUTURE ENHANCEMENTS

Several enhancements are planned for later incorporation into the system:

(1) A front-end that will compute the graph-theoretic representation from digital trafficability map data. Trafficability maps provide a plot of mobility across terrain for a given vehicle type and specified weather conditions. These maps are thus the composite of several map overlays. Other groups are currently engaged in developing the capability to automatically generate the graph-theoretic representation of trafficability maps. When software to accomplish this task becomes available, it will be incorporated into the system.

(2) The number of route intersections in an area of interest is proportional to the size of the area. The search time increases exponentially with the number of route intersections. Thus, as the size of the area increases, there is an exponential increase in the search time. For effective planning over long distances, a pratical system must have the capability to first focus its attention on major roads and then use all available route information for detailed planning in small areas.

(3) In the current implementation all graph links have a preassigned constant cost that could correspond to distance or to the time required to traverse the path corresponding to the link. An enhanced capability will enable cost to be assessed not only in terms of traversal time but also in terms of fields of fire, availability of cover, and the potential for manmade or natural obstacles.

V. REFERENCES

[1] Samet, H. The Quadtree and Related Hierarchical Data Structures, ACM Computing Surveys 16 2 (1984)

[2] Zhang, T. Y. and Suen, C. Y. A Fast Parallel Algorithm for Thinning Digital Patterns, Comm. of the ACM 27, 3 (1984) 236-239

[3] Hart, P.E., Nilsson, N. J. and Raphael, B., A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE*Trans.

VI. APPENDIXES

A. Multiple Route Finder Program

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;List of functions and calls to other functions and global variables:
;triple:
;   store-graph define-window pcopy pstore draw-arc update checklist
;   send print-answer new
;   GLOBALS USED: no-nodes-left finished startnode destnode nbrpathsfound
;               : answerlist top ptrnode1 node1
;store-graph
;   graph
;graph
;   graph
;new
;fetch
;update
;   fetch pcopy pstore new draw-arc insert-node pop print-list
;   GLOBALS USED: finished top ptrnode1
;heuristic
;     hypotenuse
;      GLOBALS USED: destnode
;draw-arc
;   fetch line
;insert-node
;   fetch store-soin pstore find-link GLOBALS USED: destnode
;find-link
;   fetch pstore prune-check
;store-soln
;   prune-check fetch pstore insert-answer
;   GLOBALS USED: nbrpathsfound answerlist
;insert-answer
;   insert-answer
;pull
;   fetch pstore  GLOBALS USED: no-nodes-left finished top
;prune-check
;   fetch prune GLOBALS USED: destnode
;prune
;   fetch pstore
;print-answer
;   pop fetch
;   GLOBALS USED: st  .i-node dest-node
;penult
;   fetch
;checklist
;   pop fetch penult push
;     GLOBALS USED: no-nodes-left finished startnode answerlist top
;print-list
;   fetch GLOBALS USED: top
;make-colors
;line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;
(defvar no-nodes-left)
(defvar finished)
(defvar startnode)
(defvar destnode)
(defvar nbrpathsfound)
(defvar answerlist)
(defvar top)
(defvar ptrnode1)
(defvar node1)
(defvar  coordinates (make-array 8))
(defvar data (make-array 8))
(defvar arr (make-array 8))
;
(defun hypotenuse (x1 y1 x2 y2)
  (sqrt (+ (^ (- x2 x1) 2) (^ (- y2 y1) 2 ) )))
;
; Heuristic estimate of distance to destination from node n based on
; Euclidean distance h(n).  The total cost at node n is f(n) = h(n) + g(n)
; where g(n) is the total cumulative cost in traversing a path from the
; start node to node n.
;
(defun heuristic (node-nbr)
  (terpri)(princ "    h[ ") (prin1 node-nbr) (princ " ] = " )
  (prin1(fix (apply 'hypotneuse
              (append (aref coordinates (1- node-nbr))
                      (aref coordinates (1- destnode )) )))))
;
;*****************************get-graph***********************************
;
;
; arguments: none
; returns: not used by calling function
;
(defun get-graph ()
  (fillarray data
             '((0 10.0 22.4 -1 -1 -1 -1 38.5)
               (10.0 0 20.0 -1 -1 -1 -1 -1)
               (22.4 20.0 0 26.0 -1 -1 -1 -1)
               (-1 -1 26.0 0 35.0 30.0 -1 -1)
               (-1 -1 -1 35.0 0 10.0 -1 -1)
               (-1 -1 -1 30.0 10.0 0 22.5 39.8)
               (-1 -1 -1 -1 -1 22.5 0 24.1)
               (38.5 -1 -1 -1 -1 39.8 24.1 0)))
  (fillarray coordinates
             '((10 10) (10 20) (30 20) (55 27) (90 29) (84 21) (72 2) (48 4)))
  (store-graph data arr))
```

```
; TRIPLE finds the N best routes between two nodes of a graph.  The
; routes are determined by constructing a best first search tree from
; the graph.  The search tree is stored in a linked list.  When terminal
; nodes are added to the list, they are inserted into the list according
; to the weight of the node.  Nodes are terminal, nonterminal, pruned, or
; destination nodes.  The next node to be searched is the top-most terminal
; node.
;
; arguments: none
; returns: none (top level function)
;
(defun triple ()
   (let (nbrpaths)
    (setq ptrnode1 nil)
    (setq answerlist nil)
    (princ
        "input node numbers of start point and of destination:  ")
    (setq startnode (read))
    (setq destnode (read))
    (terpri)
    (princ "input number of optimum paths:  ")
    (setq nbrpaths (read))
    (terpri)
    (get-graph)
;   (get-graph-using-mouse)
    (define-window)
    (setq answerlist nil)   ;initial variables
    (setq finished nil)
    (setq no-nodes-left nil)
    (setq nbrpathsfound 0)
    (setq top (new))
    (pcopy top ptrnode1)
    (pstore top nil 'downptr)
    (pstore top startnode 'nodenbr)
    (pstore top nil 'fatherptr)
    (pstore top 0 'weight)
    (pstore top 'terminal 'type)
    (pstore top (aref coordinates (sub1 startnode)) 'coordinates)
    (pstore top (heuristic startnode) 'f-weight) (terpri)
    (draw-arc top top)
    (loop1
       (if (or
             (update arr coordinates)
             (= nbrpathsfound nbrpaths))
           (checklist))
       ((eq finished t)))
    (send win-1 :set-cursorpos 0 0)
    (print-answer (reverse answerlist))))
```

```
;
;************************store-graph************************************
;
(defun store-graph
;
; function takes the coincidence matrix 'data' and generates the sparse
; matrix representation "arr".
; arguments: data - coincidence matrix represented as 1-d array of lists
;            arr  - sparse array  represented as 1-d array of lists
; returns: not used by calling function
;
   (data arr)
   (prog (m row)
        (setq m -1)
        (loop1 ((= m 7) t)
              (setq m (+ 1 m))
              (setq row (aref data m))
              (aset (graph row 0) arr m)))) ;Process mth row of data
;
;*******************************graph************************************
;
(defun graph (r n)
;
; graph recursively generates sparse list from mth sublist of input data
; stored in the global variable data.  The input representation uses -1
; to represent infinite weight (i.e., no path) and zero for the distance of
; a node to itself.  The zero and -1 values are stripped from the sparse
; list.
;
; arguments: r - a list containing mth row of coincidence matrix "data"
;            n - column number initialized to zero by store-graph
; returns: list of lists containing weight and node-nbr for each path
;
;
   (r n)
   (setq n (+ 1 n))
   (cond ((null r) nil)
        (t
         (append (cond ((<=0 (car r)) nil)
                       (t (list (list (car r) n))))
                 (graph (cdr r) n)))))

;
;
;*******************************update*********************************
;
(defun update ( arr coordinates)
;
; function inserts descendents of father node into the linked list and checks
; to make sure that a node and its grandfather node are not one and the same.
;
; arguments: arr - sparse matrix generated by store-graph
;    coordinates - 1d array of lists containing x,y coordinates of nodes
;
; returns: the Boolean variable finished
```

```
 (prog (psnew p q q0 w0)
       (setq p (pull)) ; Get new father node
       (cond
        (finished (print-list) (return finished))
        (t nil))
;      (terpri)
       (princ "father is ")
       (prin1 (fetch p 'nodenbr))
       (princ " :  ")
       (setq q (aref arr (sub1 (fetch p 'nodenbr))))
       (loop1
        (setq psnew (new))  ; Create new node
        (cond ((< > (fetch (fetch p 'fatherptr) 'nodenbr)
                    (cadar q))
               ; Then grandfather node is differentfrom son node
               (pcopy psnew ptrnode1) ; ptrnode1 points to template
               (setq q0 (cadar q))
               (pstore psnew q0 'nodenbr)
               (pstore psnew
                       (setq w0 (+ (caar q) (fetch p 'weight)))
                       'weight)
               (pstore psnew
                       (+ w0 (heuristic q0)) 'f-weight)
               (pstore psnew
                       (aref coordinates (sub1 q0)) ;print
                       'coordinates)
               (pstore psnew p 'fatherptr)
               (pstore psnew 'terminal 'type)
               (draw-arc p psnew)
               (insert-node psnew top))
              (t nil))  ; else do nothing
        (pop q)
        ((null q) (print-list) (return finished)))))
;
;
;
;*************************insert-node*********************************
;
(defun insert-node   (pin p)
;
; insert-node inserts the node <pin> into the linked list <p>. First a
; check is made to see if <pin> is the destination node. If it is, it is
; added to the solution list. If <p> points to a nonnull node, find-link
; traverses the nodes and inserts <pin> so that weight is numerically
; ordered. If <p> points to a null node, <pin> is attached to <p>.
;
; arguments: pin - pointer to node to be inserted in linked list
;            p   -  points to the head of the linked list
;
; returns: not used
;
   (pin p)
       (if (eq (fetch pin 'nodenbr) destnode)   (store-soln pin p))
       (cond ((not (null (fetch p 'downptr))) (find-link pin p))
             (t (pstore p pin 'downptr)
```

```
            (pstore pin nil 'downptr))))
;
;*****************************find-link*********************************
;
(defun find-link   (pin p)
;
; find-link traverses the list <p> and uses the weight key to insert
; <pin> in numerical order. If the end of the list is detected, <pin>
; is inserted at the tail of the list.
;
; arguments: same as insert-node above
;
; returns: not used
;
(defun find-link
   (pin p)
   (prog (ps0 ps1)
        (setq ps0 p)
        (setq ps1 (fetch p 'downptr))
        (loop1 ((null ps1) (pstore ps0 pin 'downptr)(prune-check pin p))
              ((< (fetch pin 'f-weight) (fetch ps1 'f-weight))
               (pstore pin ps1 'downptr)
               (pstore ps0 pin 'downptr)
               (prune-check pin p))
              (setq ps0 ps1)
              (setq ps1 (fetch ps1 'downptr)))))
;
;
;************************store-soln**********************************
;
(defun store-soln    (pin p)
;
; store-soln increments the variable nbrpathsfound, then sets the node type
; of <pin> to 'dest, creates a new node, copies <pin> into the new node and
; pushes the name of the new node onto the answerlist variable.
;
; arguments: same arguments as insert-node above
; returns: not used
;
   (pin p)
   (prune-check pin p)
   (cond
    ((eq 'dest (fetch pin 'type)) nil)
    (t
     (setq nbrpathsfound (1+ nbrpathsfound))
     (pstore pin 'dest 'type)
     (setq answerlist
           (if (null answerlist) ;then
                (list pin)  ;else
                (insert-answer pin (fetch pin 'f-weight) answerlist))))))
;
;
;************************insert-answer*************************************
;
(defun insert-answer (pin w lis)
```

```
;
; this function inserts the gensym name of a node onto the answerlist
; according to the value of 'weight for the node to be inserted.
; <pin> is name of node to be inserted,
; <w> is weight of <pin>,
; and <lis> is list onto which <pin> is to be inserted.
;
; arguments: pin - pointer to node to be added to list
;            w   - weight of node to be added to list
;            lis - local copy of the global variable answer-list
; returns: the new value of lis
;

   (cond
     ((null lis) (list pin))
     ((<= w (fetch (car lis) 'f-weight)) (cons pin lis))
     (t  (cons (car lis) (insert-answer pin w (cdr lis))))))

;
;
;**************************pull***********************************
;
(defun pull    ()
;
; pull locates the top-most terminal node, changes it to nonterminal and
; returns the pointer to the node. If there are no terminal nodes remaining,
; the global variable finished is set true and nil is returned.
;
; arguments: none
; returns: see comment above
;
   (prog (q0 q)
        (setq q top)
        (if (loop1 ((equal(fetch q 'type) 'terminal) nil)
                   ((null (fetch q 'downptr))
                     (setq no-nodes-left t)
                     (setq finished t))
                   (setq q0 q)
                   (setq q (fetch q0 'downptr)))
            ; then
            (return nil))
        (pstore q 'nt 'type)
         (return q)))
;
;
;**************************prune-check*****************************
;
(defun prune-check  (pin top1)
;
; prune-check checks to see if a newly inserted node has the same nodenumber as
; any other node on the linked list.  If so, the node with the larger weight is
; pruned (i.e., the node type is changed to pruned.)
;
; arguments: pin - pointer to new node being checked for pruning
;            top1 - pointer to list of nodes
```

```
; returns: not used
;
   (pin top1)
   (prog ( p n nin)
        (if (eq (setq p top1) pin)
             ; then
             (setq p (fetch p 'downptr)))
          (loop1 (setq n (fetch p 'nodenbr))
                 (setq nin (fetch pin 'nodenbr))
                 (if (and
                      (= nin  n)
                      (<> p pin)
                      (<> n destnode))
                     (prune pin p))
                 (if (eq p pin)
                     ; then
                     (setq p (fetch p 'downptr)))
                 ((null (setq p (fetch p 'downptr)))))))
;
;*************************prune*********************************
;
(defun prune  (p1 p2)
;
; prune is given pointers to two nodes on the stack with identical node
; numbers.  The node with the larger weight is marked 'pruned.
;
; arguments: see comment above
; returns: not used
;
   (p1 p2)
   (if (> (fetch p2 'f-weight) (fetch p1 'f-weight))
       ; then
      (pstore p2 'pruned 'type)
       ; else
      (pstore p1 'pruned 'type)))
;
;
;************************print-answer*************************************
;
(defun print-answer (a)
;
; print-answer first checks to see if a path was found and prints a message
; if a path was not found.  Otherwise, it prints out all paths found showing
; the route from destination back to the start node.  The answer list
; contains all the destination nodes located during the search.  This
; function traces the path up the search tree using the father pointer.
;
; arguments: a - local copy of the global variable answerlist
; returns: not used
;
  (prog (a1)
    (setq a1 (pop a))
    (cond ((null a1)
      (princ "******* NO PATH FOUND BETWEEN ")
                 (prin1 startnode)
```

```
                    (princ " AND ")
                    (prin1 destnode)
                    (return nil)))
      (loop1
        (terpri)
        (princ "distance is  ")(prin1 (fix (fetch a1 'weight)))
        (princ ". Path from destination back to start is:")
          (terpri)
          (prin1 (fetch a1 'nodenbr))
        (loop1
          ((null (fetch a1 'fatherptr)))
          (prin1 '-)
          (setq a1 (fetch a1 'fatherptr))
          (prin1 (fetch a1 'nodenbr)) )
        ((null (setq a1 (pop a))) (terpri)))))
;
;
;**************************penult*********************************
;
(defun penult (a1 firstnode father)
;
; penult traverses a linked list and returns the node number of the
; penultimate node.  The first argument <a1> is the pointer to the head of
; the list.  The second argument <firstnode> normally has the value of
; <startnode> which is the top of the search tree.  If some other pointer
; is used, then this function will return the node prior to <firstnode>.
; The third argument <father> is the pointer to the next node on the tree.
; For the search tree the next node pointer is 'fatherptr, while for the
; linked stack the pointer is 'downptr.  If the list is null or has only
; one node, then nil is returned.
;
; arguments     a1 - pointer to the head of the list
;          first-node - see comment above
;             father - see comment above
;
;  returns: see comment above
;
(defun penult (a1 firstnode father)
  (loop1
    ((null a1) nil)
    ((= firstnode (fetch a1 'nodenbr)) nil)
    ((= firstnode (fetch (fetch a1 father) 'nodenbr)) a1 )
    (setq a1 (fetch a1 father))))
.
.
.
.
*****************************checklist*******************************
.
(defun checklist ()
.
 checklist is called only if nbrpathsfound equals or exceeds the required
 number.  The function tests to see if the search is finished in a two-step
 process.  First, the answer list is scanned to find the largest weight on
 the answer list.  This value is compared against the weight of all terminal
```

```
; nodes on the stack.  If any terminal node has a smaller weight, then the
; search is not finished, and finished is not set to true.  If finished is
; set to true, then the program checks to see if any two paths on the
; answerlist have the same penultimate nodes.  This condition occurs when
; two separate paths initially have the same route but diverge and come
; together at the destination node.  The pruner does not detect this case
; since the node has already been marked as a destination node.
;
; argument: none
; returns:
;
   (prog (p a w lis)
        (setq lis answerlist)
        (setq p top)
        (setq a (pop lis))
        (if (null a)  (return nil))
        (setq w (fetch a 'f-weight))
        (loop1 ((eq t no-nodes-left))
              (setq a (pop lis))
              ((null a))
              (if (> (fetch a 'f-weight) w)
                  ; then
                  (setq w (fetch a 'f-weight))))
        (setq finished t)
        (loop1 ((eq t no-nodes-left))
              (if (and (equal (fetch p 'type) 'terminal)
                       (< (fetch p 'f-weight) w))
                  ; then
                  (setq finished nil))
              (setq p (fetch p 'downptr))
              ((null p)))

        (cond (finished
                 (print(setq w answerlist))
                 (setq lis nil);initialize list of penultimate node numbers
                 (setq answerlist nil)
                 (loop1
                   ((null w))
                   (cond ((member
                             (setq p (penult (car w) startnode 'fatherptr))
                             lis)
                          (pop w))
                         (t  (push  p lis)
                             (push (pop w) answerlist)))))) ))


;
;**************************print-list*********************************
;
(defun print-list  ()
;
; print-list is given the top of the linked list used as the best-first
; search stack.  For each node, the node number is printed, followed
; immediately by node type unless the node is a terminal node.  Codes for
; terminal types are as follows:
```

```
; d -- destination, p -- pruned, n -- non-terminal and no letter indicates a
; terminal node.  Nodes are separated by hyphens.
;
; arguments: none
; returns: not used
;
   (prog (p q)
         (setq p top)
         (setq q (fetch p 'downptr))
         (prin1 (fetch p 'nodenbr))
         (cond ((equal (fetch p 'type) 'pruned) (prin1 'p))
               ((equal (fetch p 'type) 'nt) (prin1 'n))
               ((equal (fetch p 'type) 'dest) (prin1 'd))
               (t nil))
         (loop1 ((null q))
                (setq p q)
                (setq q (fetch p 'downptr))
                (prin1 '-)
                (prin1 (fetch p 'nodenbr))
                (cond ((equal (fetch p 'type) 'pruned)
                       (prin1 'p))
                      ((equal (fetch p 'type) 'nt)
                       (prin1 'n))
                      ((equal (fetch p 'type) 'dest)
                       (prin1 'd))
                      (t nil)))
         (terpri)))
```

```
;;; -*- Syntax: Zetalisp; Package: USER; Base: 10; Mode: LISP -*-
; Following function is common to both property list
; and defstruct storage methods.
;
(defun new ()
  (intern(gensym)))
;
(defvar node)  ; common to both structures

;
;  Access functions for defstructures storage:

(defstruct (node)
        (nodenbr )
        (fatherptr )
        (downptr )
        (weight )
          (f-weight)
        (type )
        (coordinates ))
;
;*********************************pcopy*********************************
;
; This function makes instance of the defstruct.  The variable old is
; kept for compatibility with the equivalent prop-list function.
(defun pcopy (new old)
;
; arguments: new - pointer to copy of defstruct
;            old - pointer to original defstruct
; returns: old
;
(defun pcopy (new old)
  (set new (make-node))
  old)
;
;*******************************fetch*****************************************
;
(defun fetch (name prop)
;Compiler generates a spurious message that the function prop is not defined.
;
; arguments: name - pointer to defstruct
;              prop - defstruct slot name
; returns: slot value
;
  (iff (null name) then nil ;iff defined in zlib.lisp
       else
       (fset 'prop (fsymeval prop))
       (prop (eval name))))
;
;*******************************pstore****************************************
;
(defun pstore (name value prop)
;
; arguments: name - pointer to defstruct
;              value - defstruct slot value
```

```
;                prop - defstruct slot name
; returns: slot value
;
  (eval `(alter-node ',(eval name) ,prop ',value)))
;
;*****************************************************************************;
;
; access functions for property list storage:
; Inclosure of function in |#  <function>  #| prevents compilation unless
; a shift-control-c operation is performed inside
; the block.  The functions have the same arguments and returns as the
; defstruct equivalents defined above.
;
|#
(defvar ptrnode1)
(defvar ptrnode1 'node)
(defprop node
        ((nodenbr val1)
         (fatherptr val2)
         (downptr nil)
         (weight val4)
           (f-weight 0)
         (type val5)
         (coordinates (x y)))
        stackptr)
;
; pstore has the same arguments as the LISP function put.  However the
; third argument is the key for the association list stored under the
; property 'stackptr of the atom specified by the first argument.
;
(defun pcopy (new old)
  (remprop new 'stackptr)
  (putprop new (get old 'stackptr) 'stackptr))
;
(defun pstore (atom value key)
  (putprop atom
           (subst (cons key (list value))
                  (cons key (list (fetch atom key)))
                  (get atom 'stackptr))
           'stackptr))
;
; fetch returns the value associated with <key>.  The association list
; is stored under the property 'stackptr on the variable <nameptr>.

;
(defun fetch (nameptr key) (cadr (assoc key (get nameptr 'stackptr))))
;
#|
```

B. Graph Generation Using Mouse

```
;;; -*- Syntax: Zetalisp; Package: USER; Base: 10; Mode: LISP -*-
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; List of functions and functions they call:
;  get-graph-using-mouse
;         trace-arc-using-mouse
;         Global Variables: coordinates *max-no-nodes* x-arc-pts y-arc-pts
;                           *j* *k*
; trace-arc-using-mouse
;         old-node-p plot-solid-circle-draw-segment
;         Global Variables: arr *max-no-nodes* x-temp-list y-temp-list nbr-pts
;                           *j* *k* px py *next-unused-node* coordinates red
; plot-solid-circle
;         Global Variables: win-1
; old-node-prep
;         Global Variables: coordinates *max-no-nodes*
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
(defvar *max-no-nodes* 8)
(defvar  coordinates)
(defvar data (make-array  *max-no-nodes*))
(defvar arr)
(defvar x-arc-pts)
(defvar y-arc-pts)
(defvar *next-unused-node*)
(defvar *j*)
(defvar *k* 0)
(defvar nbr-pts)
(defvar x-temp-list nil)
(defvar y-temp-list nil)
;
;
;
***************************old-nodes-p********************************
;
(defun old-node-p (x y) ;return nil only if node is new
; function checks if node at (x y) corresponds to previously tagged node
; within a precision of 10 pixels for each axis
;
; arguments: x - horizontal coordinate of new node
;            y - vertical coordinate of new node
;
; returns: nil if new node, node number if old node
;
 (let ((n 0))
  (loop1
   ((null (aref coordinates n)) nil )
   ((and
       (< (abs (- x (car (aref coordinates n)))) 10)
       (< (abs (- y (cadr (aref coordinates n))))10)) n  )
```

```
    ((= (sub1 *max-no-nodes*) n)
;     (setf (aref coordinates n) (list x y))
      nil)
     (incf n))))
;
;
(defun plot-solid-circle ( x y value color)
;
; Function prints solid circle, prints node number in circle and repositions
; cursor at center of circle
;
; arguments x y - coordinates of circle center
;           value - node number
;           color - color of circle
; returns: not used
;
  (let (x1 y1)
    (multiple-value (x1 y1) (send win-1 :read-cursorpos))
    (send win-1 :draw-filled-in-circle x y 10 (color:sc-fill-alu color -1))
    (send win-1 :set-cursorpos (- x 3) (- y 4))
    (prin1 value win-1)
    (send win-1 :set-cursorpos x1 y1) ))
;
;
;
;
**********trace-arc-using-mouse**********
;
(defun trace-arc-using-mouse ()
;
; Function calls draw-segment which, fits spline curves to a road segment.
; One or more segments constitutes an arc with a solid circle drawn at each
; end of the arc.  When draw-segment returns nil, the last arc has been
; drawn.  Otherwise draw-segment returns a list that is stored in curve.
; The first value is the traversal distance of the segment, and the second
; value is used to test if segment is the last segment of an arc.  If it is,
; then the node at the end of the arc is drawn if it was not previously
; drawn.  If the segment is the first segment of an arc and the function
; old-node-p returns nil, then the node at the beginning of the arc is drawn.
;
; arguments: none
; returns: see comment above
;
  (let ((segment-nbr 0)  (cum-distance 0)
        curve (continue t))
    (setq arr (make-array  *max-no-nodes*))
    (setq x-temp-list nil)
    (setq y-temp-list nil)
    (loop!
      (setq px (make-array 20 :type art-16b :fill-pointer 0));for storing x
      (setq py (make-array 20 :type art-16b))       ;and y mouse coordinate
      ((null (print (setq curve (draw-segment)))) (setq continue nil))
      (setq cum-distance ( + cum-distance (car curve)))
      (setq nbr-pts (- (array-leader px 0) 1))
      ((    1 (second curve))                ;is this the last segment of the arc?
```

```
    (print "last point of arc")              ;yes
    ;draw node at end of arc if not previously drawn
    (iff
        (null (old-node-p (aref px nbr-pts) (aref py nbr-pts)))
        then
        (setq *k* *next-unused-node*)              ;process new node
        (plot-solid-circle (aref px nbr-pts)(aref py nbr-pts) *k* red)
        (incf *next-unused-node*);
        (setf (aref coordinates *k*)
            (list (aref px nbr-pts) (aref py nbr-pts)))) )
   (iff (not (null px)) then (print (push px x-temp-list)))
   (iff (not (null py)) then (push py y-temp-list))
   ;draw node at beginning of arc if not previously drawn
   (iff
       (and
        (=0 segment-nbr)  ;first segment of arc   AND
        (null (old-node-p (aref px 0) (aref py 0))))
       then
       (setq *j* *next-unused-node*)              ;process new node
       (plot-solid-circle (aref px 0)(aref py 0) *j* red)
       (incf *next-unused-node*);
       (setf (aref coordinates *j*) (list (aref px 0) (aref py 0))) )
   (iff
       (and
         (=0 (print segment-nbr))  ;first segment of arc   AND
         (=0 *next-unused-node*))  ;first arc of graph
       then
       (setf (aref coordinates 0) (list (aref px 0) (aref py 0)))
       (plot-solid-circle (aref px 0)(aref py 0) 0 red)
       (incf *next-unused-node*)
       (setq *j* 0))
   (incf segment-nbr)
   (setq *j* *k*))                            ;end of loop
  (print x-temp-list)
  (iff continue
        then  (princ "*** j k  ") (prin1 *j*)(princ " ")(prin1 *k*)
        (print x-temp-list)
        (setf (aref x-arc-pts *j* *k*) x-temp-list)
        (setf (aref x-arc-pts *k* *j*) x-temp-list)  (setq x-temp-list nil)
        (setf (aref y-arc-pts *j* *k*) y-temp-list)
        (setf (aref y-arc-pts *k* *j*) y-temp-list)  (setq y-temp-list nil)
        (setf (aref arr *k*)
            (append (list(list cum-distance *j*)) (aref arr *k*)))
        (setf (aref arr *j*)
            (append (list(list cum-distance *k*)) (aref arr *j*)))) ))
;
;
;
*******************get-graph-using-mouse*****************************
;
(defun get-graph-using-mouse ()
;
; This function is the highest level function in the mouse-graph module.
; The MRF module calls either get-graph or get-graph-using-mouse, depending
; on whether graph data have previously been digitized or are to be
```

```
; obtained using the mouse to extract routes from a cross-country mobility
; map displayed on the color monitor.  This function first initializes
; several arrays.  The function define-color-window is called and a color
; window is created.  The function trace-arc-using-mouse is in a loop that
; continues until nil is returned by the function.
;
; arguments: none
; returns: not used
;
 (setq *j* 0)
 (setq *k* 0)
 (setq coordinates (make-array *max-no-nodes*))
   (setq x-arc-pts (make-array (list *max-no-nodes* *max-no-nodes*)))
   (setq y-arc-pts (make-array (list *max-no-nodes* *max-no-nodes*)))
   (setq *next-unused-node* 0)
   (loop1
     ((null (trace-arc-using-mouse)))  )
   'finished)
```

C. Screen Control Functions

```
;;; -*- Syntax: Zetalisp; Package: USER; Base: 10; Mode: Lisp -*-
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; List of functions and functions they call:
; draw-arc
;         color-draw-arc bw-draw-arc
;         Global variables: *screen-type*
; color-draw-arc
;         fetch line
;         Global variables: win-1 red
; line
;         Global variables: win-1
; bw-draw-arc
;         fetch
;         Global variables: win-1
; define-window
;         make-colors  define-color-window define-bw-window
;         Global variables: *screen win-1 *flag*
; define-bw-window
;         Global variables: win-1 *middle* *top* *right* *bottom*
; define-color-window
;         make-colors
;         Global variables: win-1
; make-colors
;         Global variables: black red green orange blue white *flag*
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
(defvar *middle* 544)
(defvar *top* 0)
(defvar *right* 1088)
(defvar *bottom* 736)
(defvar win-1)
(defvar *screen-type* 'color) ;default screen type is color
(defvar *flag* nil)                    ; set to t when define-window is called
(defvar black     0)
(defvar red       1)
(defvar green     2)
(defvar orange    3)
(defvar blue      4)
(defvar white     5)
(defvar *y-offset*)
(defvar *x-scaler*)
(defvar *y-scaler*)
;
;******************************define-bw-window***************************
;
; arguments: none
; returns: not-used
;
(defun define-bw-window ()
```

```
    (setq win-1 (tv:make-window 'tv:window
                        :left (add1 *middle*)
                        :top *top*
                        :right *right*
                        :bottom *bottom*
                        :expose-p t))
    (setq *y-offset*
          (* (// (fix (* 0.8 (send win-1 :height))) 100) 100))
    (setq *x-scaler* (// *y-offset* 100))
    (setq *y-scaler* (* -1 *x-scaler*))
    (setq *flag* t) )
;
(defvar win-2)
(defun define-bw-windows ()
  (let (x0 y0 xn yn x1 y1)
    (multiple-value (x0 y0 xn yn)
          (send tv:main-screen :edges))
    (setq x1 (+ x0 (// (- xn x0) 2)))
    (setq y1 (+ y0 (fix (* (- yn y0) .6))))
    (tv:make-window 'tv:lisp-listener
                        :superior tv:main-screen
                        :edges (list x0 y0 x1 y1)
                              :expose-p t)
    (tv:make-window 'tv:lisp-listener
                        :superior tv:main-screen
                        :edges (list (1+ x1) y0 xn y1)
                              :expose-p t)
    (setq win-1 (tv:make-window 'tv:window
                        :left x0
                        :top (1+ y1)
                        :right x1
                        :bottom yn
                        :expose-p t))
    (setq win-2 (tv:make-window 'tv:window
                        :left (1+ x1)
                        :top (1+ y1)
                        .right xn
                        :bottom yn
                        :expose-p t))
    (setq *y-offset*
          (* (// (fix (* 0.8 (send win-1 :inside-height))) 100) 100))
    (setq *x-scaler* (// (send win-1 :inside-width) 100))
    (setq *y-scaler* (* -1 *x-scaler*))
    (setq *flag* t)))

;
(Defun make-colors ()
;
; Modifies the lookup table for addresses 0 through 5
;
;**********make-colors**************************************
;
; arguments: none
; returns: not used
;
```

```
  (send color:color-screen :write-color-map black 0 0 0 0)
  (send color:color-screen :write-color-map red 1023 0 0 0)
  (send color:color-screen :write-color-map green 0 1023 0 0)
  (send color:color-screen :write-color-map orange 850 261 104 0)
  (send color:color-screen :write-color-map blue 0 0 1023 0)
  (send color:color-screen :write-color-map white 1023 1023 1023 0)  )
;
;**********************define-color-window************************
;
; arguments: none
; returns: not used
;
(defun define-color-window ()
  (setq win-1 (tv:make-window 'tv:window
                              :superior color:color-screen
                              :borders 10
                              :save-bits t
                              :edges '(128 50 1151 1000)
                              :blinker-p nil
                              :font-map '(fonts:cptfont fonts:hl7)
                              :char-aluf tv:alu-xor
                              ))
  (make-colors)
  (send win-1 ':expose)
  (send win-1 ':clear-input)
  (send win-1 ':clear-window)
  (setq *flag* t))
;
;
;************************define-window**************************
;
(defun define-window ()
;
; function first tests *flag* to see if window has already been created.
; function then tests to see if window is to be color or black and white
;
;
; arguments: none
; returns: not used
;
  (cond
    ((eq *flag* t) t)
    ((eq *screen-type* 'color) (define-color-window))
    (t (define-bw-window))))
;
;
;********************bw-draw-arc*****************************************
;
(defun bw-draw-arc  (p1 p2)
;
; function uses pointers to access node numbers and coordinates in order
; to draw a line between the two nodes and solid circles centered at the
; node positions and with the node numbers printed in the circles.
;
; arguments: p1 p2 - pointers to nodes
```

```
; returns: not used
;
 (let((x1 (* *x-scaler* (car (fetch p1 'coordinates)) ))
      (y1 (+ *y-offset* (* *y-scaler* (cadr (fetch p1 'coordinates)))))
      (x2 (* *x-scaler* (car (fetch p2 'coordinates) )))
      (y2 (+ *y-offset* (* *y-scaler* (cadr (fetch p2 'coordinates))))))
   (send win-1 :draw-line
         x1 y1 x2 y2
         tv:alu-ior)
   (send win-1 :draw-filled-in-circle x2 y2 10 tv:alu-ior)
   (send win-1 :set-cursorpos x2 (+ 15 y2))
   (prin1 (fetch p2 'nodenbr) win-1)))
;
;
;*************************line*****************************************
;
(defun line (win-1 xstart ystart xend yend color)
;
; draws line between two points.
;
; arguments: xstart ystart - coordinates of first point
;              xend yend   - coordinates of second point
;                   color  - color of line to be printed
; returns: not used
;
  (send win-1 ':draw-line xstart ystart xend yend
        (color:sc-fill-alu (eval color) -1))) ;
;
;
;
;**********************color-draw-arc*************************************
;
(defun color-draw-arc  (p1 p2)
;
; function uses pointers to access node numbers and coordinates in order
; to draw a line on the color monitor between the two nodes and solid
; circles centered at the node positions and with the node numbers printed in
; the circles.
;
; arguments: p1 p2 - pointers to nodes
; returns: not used
;
  (let ((x1 (* 10 (car (fetch p1 'coordinates)) ))
        (y1 ( + 800 (* -10 (cadr (fetch p1 'coordinates) ))))
        (x2 (* 10 (car (fetch p2 'coordinates) )))
        (y2 ( + 800 (* -10 (cadr (fetch p2 'coordinates) )))))
      (line win-1 x1 y1 x2 y2 red)
   (send win-1 :draw-filled-in-circle x2 y2 10 tv:alu-ior)
   (send win-1 :draw-filled-in-circle x2 y2 10 (color:sc-fill-alu red -1))
   (send win-1 :set-cursorpos (- x2 3)  (- y2 4))
   (prin1 (fetch p2 'nodenbr) win-1)))
;
;
;
;**********************draw-arc*******************************************
```

```
;
(defun draw-arc (p1 p2)
;
; function tests if the screen type is color or black  white and calls the
; corresponding function for drawing an arc between the nodes specified by
; p1 and p2
;
; arguments: p1 p2 - pointers to nodes
; returns: not used
;
   (iff (eq *screen-type* 'color)
    then (color-draw-arc p1 p2)
    else (bw-draw-arc p1 p2) ))
```

D. Line- and Curve-drawing Functions

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10; -*-
; Several functions were previously written by W.W. Seemuller with extensive
; modifications and name changes by J.R. Benton
;
(defvar px (make-array 20 :type art-16b :fill-pointer 0))  ;for storing x and
(defvar py (make-array 20 :type art-16b))                  ;y mouse coordinates
;
;*******************************draw-segment*********************************
;
(defun draw-segment ()
;
; draws lines on window 1 with function 'connect-points' - left click
; puts point, middle click indicates point is last point.  Function
; clears the straight line curve before a drawing cubic spline curve
; through the points with 'draw-cubic-spline'. The number of points used
; to generate the curve is returned.
;
; arguments: none
; returns: not used
;
 (let (distance)
  (iff (null *flag*) then (define-window) (setq *flag* t))
  (iff
   (null
      (setq distance (traverse-road win-1 px py)))        ;put initial points
   then nil
   else
   (iff (neq 0 distance)
        then
        (send win-1 :draw-curve px py(array-leader px 0)tv:alu-xor); erase
        (send win-1 :draw-cubic-spline px py 10))  ;draw cubic spline curve
   (list distance (array-leader px 0)))))  ;return number of points found
;
;
;*****************************traverse-road*********************************
;
;
(defun traverse-road (win-1 x-cor y-cor)
;
; This function calls connect-points-with-line and uses the Pythagorean
; theorem to compute the length of each straight-line section defined by the
; arrays x-cor and y-cor.  The lengths are summed to get an approximate
; value for the length of the spline curve fitted to the sequence of points.
;
; arguments: win-1 - window name used by all modules
;            x-cor - array of x coordinates computed by connect-points-with-
;                    line
;            y-cor - array of y coordinates computed by connect-points-with-
;                    line
;
 (let ((w 0))
  (iff (null
```

```
        (connect-points-with-line win-1 x-cor y-cor))
      then nil
      else
      (loop for n from 0 to (- (array-leader x-cor 0) 2) do
        (setq w
              (+ w
                (sqrt (+
                        (^ (- (aref x-cor n) (aref x-cor (add1 n))) 2)
                        (^ (- (aref y-cor n) (aref y-cor (add1 n))) 2)
                        )))))
      w) ))

;
;*****************************display-curve*********************************
;
;(defun display-curve (px py)      ; draw the cubic spline
;  (send win-1 :draw-cubic-spline px py 10))
;
;
;*****************************clear-window**********************************
;
(defun clear-win ()
  (send win-1 :clear-window))
;
;******************connects-points-with-line*******************************
;
;;;  (connect-points-with-line window x-cor y-cor &optional alu)
;;;
;;;  arguments:  window - exposed window of flavor tv:window
;;;              x-cor - one dimensional array
;;;              y-cor - one dimensional array
;;;              alu - optional color alu
;;;
;;;  returns:  number of points moused or nil if <R1> is clicked
;;;
;;;  This function places points on a window with the mouse and draws lines
;;;  between them while storing the point coordinates in two one-dimensional
;;;  arrays.  "Window" is the exposed window where the points are placed,
;;;  "x-cor" and "y-cor" are the two arrays where the relative window
;;;  coordinates are stored.  Clicking left once places points, clicking
;;;  middle places the last point and exits.  The number of points is stored
;;;  in the fill-pointer for array "x-cor" (so x-cor must be created to
;;;  handle this) and the function returns the number of points.
;;;  If "alu" is not passed, lines are drawn with the tv:alu-xor option
;;;  and can be cleared by sending a :draw-curve message as:
;;;
;;;    (send window :draw-curve x-cor y-cor (array-leader x-cor 0) tv:alu-xor)
;;;
;;;                                        WWS - 05/17/85
;;;   Modified by JB to exit without processing points if <R1> is clicked.


(defun connect-points-with-line (window x-cor y-cor &optional alu)
  (tv:mouse-set-sheet window)
  (let (button x y first-point
```

```
      (line-alu (cond (alu) (t tv:alu-xor))))
(multiple-value (button x y ) (get-mouse window))
(iff (neq button 4)
      then
      (aset x x-cor 0)
      (aset y y-cor 0)
      (store-array-leader 1 x-cor 0)
      (send window :draw-point x y line-alu)
      (do ((index 1))
          ((eq button 2)
           (array-leader x-cor 0))
        (multiple-value (button x y ) (get-mouse window))
        (cond ((not
                  (and (eq x (aref x-cor (sub1 index)));throw away
                       (eq y (aref y-cor (sub1 index))))) ;duplicate points
              (aset x x-cor index)
              (aset y y-cor index)
              (store-array-leader (add1 index) x-cor 0)
              (cond ((null first-point)
                     (send window :draw-point (aref x-cor 0)(aref y-cor 0)
                           line-alu)
                     (setq first-point t)))
              (send window :draw-line (aref x-cor (sub1 index))
                    (aref y-cor (sub1 index)) x y line-alu nil)
              (setq index (add1 index)))))
      else
      (tv:mouse-set-sheet (send terminal-io :superior))
      nil) ))
```

E. Special Macros

```
; Macros used for the automated route finder for multiple tank columns
;
;
;*******************************iff*************************************
;
(macro iff (s ignore)
;
; iff is the standard
;            (if form1 then form2 form3 ... else form_a  form_b ...)
; iff requires that then be present.
;
     (append (cons 'cond
                   (list (cons (cadr s)
                               (do (then-forms (x (cdddr s) (cdr x)))
                                   ((or (null x)
                                        (eq (car x) 'else))
                                    then-forms)
                                  (setq then-forms
                                        (append then-forms
                                                (list (car x))))))))
             (cond ((null (cdr (member 'else s))) '((t nil)))
                   (t (list (cons 't (cdr (member 'else s))))))))
;
;*******************************loop1***********************************
;
;
(macro loop1 (s ignore)
;
; The macro loop1 has syntax similar to cond except that it loops instead
; of falling through if no predicate evaluates to t.  The syntax is shown
; below:
;   (loop1
;         ((fcn_1) form_2 form_3 form_4 ...)
;         (fcn_a)
;           ..
;         ((fcn_A) form_B ...))
; where fcn_x indicates a function and form_x indicates an arbitrary
; expression.
; If the macro sees only a single parenthesis at the start of a line, it does
; not treat the function that follows as a predicate but simply executes the
; function and drops to the next line.  Alternatively, ((setq a t) 'exit)
; will result in the loop being exited with a value of 'exit returned.
;
     (let ((result '(do () (nil))))
        (dolist (form (cdr s))
          (cond ((listp (car form))
                 (setq
                   result
                   (append
                     result
                     `((cond (,(car form)
                              ,@(reverse (cdr (reverse (cdr form))))
```

```
                    (return ,(car (last (cdr form))))))))))
          (t (setq result (append result (list form))))))
result))
```

F. Discussion of Avenues of Approach and Obstacles*

Obstacles are natural and artificial terrain features that stop, impede, or divert movement of troops, equipment, or weapons, and therefore are a key component in determining mobolity. Analysis of obstacles is an important element of terrain analysis. Examples of natural obstacles are rivers, streams, lakes, swamps, marshes, cliffs, steep slopes (greater than 45 degrees), dense woods, jungles, deserts, mountains, unstable soil, e.g., peat, muck, and sand dunes. Man-made obstacles include minefields, craters, antitank ditches, trenches, abatis, roadblocks, built-up areas, flooded areas, nuclear/chemical/biological-contaminated areas, extensive rubble areas, and blow-down areas.

Avenues of approach are unimpeded routes that can be traversed to the desired destination. Wide, covered valleys, open-forested ridges, and ridge slopes below the crest of a ridge offer features usable as avenues. Since these avenues are free of obstacles, they do not resist reasonable cross-country movements. The surface configuration and materials are smooth and friendly to movement in any direction, and consist of high density soils without major streams or waterways. The ideal avenue of approach enables troops, weapons, and equipment to maneuver through an area to the to the objective without delay. A regiment requires an approximately 5 to 10 km wide valley or ridge for an avenue of approach.

* The following material was extracted from an unpublished paper by Prof. Olin Mintzer

END

FEB.

1988

DTIC